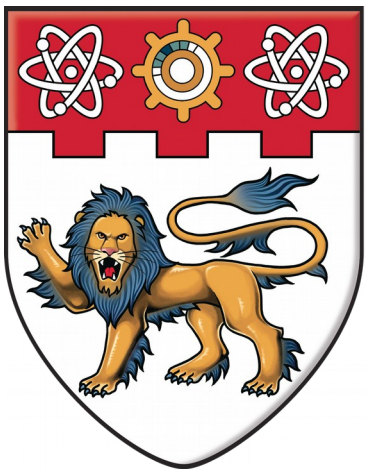

NANYANG TECHNOLOGICAL UNIVERSITY

CZ2005 Operating Systems

Experiment 3 – Report

Kamil Florowski – N1902302G

Lab Group:

SSR2

Contents

Introduction

In Lab 3 of CZ2005 Operating System course, it is required to implement virtual memory for NachOS. Components that are implemented consist of procedure of getting a physical frame for a page from Inverted Page Table, inserting an entry to TLB and eventually handling the case of missing page inside IPT (*least recently used page* replacement algorithm used).

Standard address translation scheme might be slow due to the memory access time (every instruction takes 2 x memory access time – one for page table access and one for instruction access). Therefore, we can reduce the look-up time using hardware cache called translation look-aside buffers (TLBs). Each TLB entry consist of two parts - key and value so that when the associative memory is presented with the item, it can be compared with all the keys simultaneously. TLB table must be keep small to perform in expected manner so entries are replaced while table is full (FIFO basis).

If there is no matching entry in TLB table then appropriate function is called and searching of IPT is done. It might be the case that there is no matching page in IPT either so then execution is proceeded with swapping pages to and from the disk (store).

Two main data components/structures used frequently in assignment are: Inverted Page Table ant TBL table.

Implementation of IPT's entry is represented by MemoryTable class and can be found in *machine/ipt.h:*

```
class MemoryTable {
 public:
  MemoryTable(void);
  ~MemoryTable(void);

  bool valid;                           // if frame is valid (being used)
  SpaceId pid;                          // pid of frame owner
  int vPage;                            // corresponding virtual page
  bool dirty;                           // if needs to be saved
  int TLBentry;                         // corresponding TLB entry
  int lastUsed;                 // used to see record last used tick
  OpenFile *swapPtr;                    // file to swap to
};
```

Implementation of TLB's entry is represented by TranslationEntry class and can be found in machine/translate.h:

```
class TranslationEntry {
  public:
    int virtualPage;    // The page number in virtual memory.
    int physicalPage;   // The page number in real memory (relative to the
                        //  start of "mainMemory"
    bool valid;         // If this bit is set, the translation is ignored.
                        // (In other words, the entry hasn't been initialized.)
    bool readOnly;      // If this bit is set, the user program is not allowed
                        // to modify the contents of the page.
    bool use;           // This bit is set by the hardware every time the
                        // page is referenced or modified.
    bool dirty;         // This bit is set by the hardware every time the
                        // page is modified.
};
```

All required implementation of the functions can be found under section below.

Implementation of the functions

int VpnToPhyPage(int vpn)

```
//----------------------------------------------------------------------
// VpnToPhyPage
//      Gets a phyPage for a vpn, if exists in ipt.
//----------------------------------------------------------------------
int VpnToPhyPage(int vpn)
{
  // your code here to get a physical frame for page vpn
  // you can refer to PageOutPageIn(int vpn) to see how an entry was created in ipt
  // loop through all the records in IPT
  // NumPhysPages - global variable, defined in machine.h
  for(int i=0; i<NumPhysPages; i++){
    // check conditions for needed page
    if(memoryTable[i].valid
    && memoryTable[i].pid==currentThread->pid
    && memoryTable[i].vPage==vpn)
    {
      return i; // return physical page for a VPN from IPT
    }
  }
  // if no entry found ----> return -1
  return -1;
}
```

int VpnToPhyPage(int vpn) - this function implemented in *tlb.cc* takes as a parameter page number and is looking for corresponding frame. The 'for' loop iterates through all the entries in IPT and checks whether there is frame with valid bit set to 'true', with the same page number and current process id. If appropriate frame is found, then it can be returned and eventually inserted into TLB as combination of vpn and phyPage. If loop terminates and no frame was found, function returns '-1' value what indicates that further step is needed.

void InsertToTLB(int vpn, int phyPage)

```
//----------------------------------------------------------------------
// InsertToTLB
//      Put a vpn/phyPage combination into the TLB. If TLB is full, use FIFO
//    replacement
//----------------------------------------------------------------------
// FIFOPointer pointer to the 'oldest' entry in TLB
static int FIFOPointer = 0;
void InsertToTLB(int vpn, int phyPage)
{
  int i = 0; //entry in the TLB

  //your code to find an empty in TLB or to replace the oldest entry if TLB is full
  for(;i<TLBSize;i++){
    if(!machine->tlb[i].valid){
      break;
    }
  }
  // if all entries are valid ---> record used on FIFO basis
  if(i == TLBSize){
    i = FIFOPointer;
  }
  // setting FIFOPointer to the next oldest entry
  FIFOPointer = (i+1) % TLBSize;

  ...
}
```

void InsertToTLB(int vpn, int phyPage) – above function implemented in *tlb.cc* takes two parameters: numbers of the page and frame. The combination of these two values is then inserted into TLB table. The 'for' loop iterates through all the entries inside TLB looking for invalid entry (entry not in use or terminated already). If there is entry meeting such a condition, loop is halted and 'i' value indicates row in TLB to be inserted into. If loop finishes and invalid entry is not found then FIFO policy is used (oldest inserted record is the one to be replaced). After all that, static variable is updated. In not shown fragment of the code dirty data is copied to the memoryTable and lastUsed property of memoryTable instance is set to current tick (this fact will be used in the "Analysis test program" part of the report and LRU algorithm implementation below). At the last step in the function, 'TLB misses' counter is increased what can be used for performance evaluation.

int lruAlgorithm(void)

```
1  //----------------------------------------------------------------------
2  // lruAlgorithm
3  //      Determine where a vpn should go in phymem, and therefore what
4  // should be paged out. This lru algorithm is the one discussed in the
5  // lectures.
6  //----------------------------------------------------------------------
7  int lruAlgorithm(void)
8  {
9    //your code here to find the physical frame that should be freed
10   //according to the LRU algorithm.
11   //Assumption: NumPhysPages>=1
12   int phyPage;
13   int lastUsedValue = memoryTable[0].lastUsed;
14
15   for(int i=0; i<NumPhysPages; i++){
16     // checking if entry is valid
17     if(!memoryTable[i].valid){
18       return i;
19     }
20     // checking if lastUsed value is lesser than min
21     if(memoryTable[i].lastUsed<lastUsedValue){
22       lastUsedValue = memoryTable[i].lastUsed;
23       phyPage = i;
24     }
25   }
26   return phyPage;
27 }
```

int lruAlgorithm(void) – above function implemented in *tlb.cc* takes no arguments but returns integer – number of the physical page which needs to be replaced. LastUsedValue is the variable which stores the minimum value of the ticks (time) when page was accessed (initially set to lastUsed property of the first frame). Then the algorithm loop iterates through IPT looking for invalid entry (which is good to be replaced immediately). If there is no such an entry – this is the moment when LRU policy comes up: there is a check in line 21 whether the lastUsed value of the current page is lesser than the minimum value – lastUsedValue. As a result if no invalid entry will be found (return from line 18 is not called), algorithm reaches line 26 and returns page with the lowest number of ticks – page which has not been used for longest period of time.

Notice: for code clarity DEBUG lines were removed from the code snippets

Test program output analysis

tick	vpn	pid	IPT[0]	IPT[1]	IPT[2]	IPT[3]	TLB[0]	TLB[1]	TLB[2]	Page Out
10	0	0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0	0,0,0	0,0,0	
13	9	0	0,0,12,1	0,0,0,0	0,0,0,0	0,0,0,0	0,0,1	0,0,0	0,0,0	
15	26	0	0,0,12,1	0,9,15,1	0,0,0,0	0,0,0,0	0,0,1	9,1,1	0,0,0	
20	1	0	0,0,12,1	0,9,19,1	0,26,17,1	0,0,0,0	0,0,1	9,1,1	26,2,1	
26	0	0	0,0,12,1	0,9,25,1	0,26,17,1	0,1,22,1	1,3,1	9,1,1	26,2,1	
28	10	0	0,0,28,1	0,9,25,1	0,26,17,1	0,1,22,1	1,3,1	0,1,1	26,2,1	26
41	9	0	0,0,40,1	0,9,25,1	0,10,28,1	0,1,22,1	1,3,1	0,0,1	10,2,1	
42	26	0	0,0,40,1	0,9,42,1	0,10,28,1	0,1,22,1	9,1,1	0,0,1	10,2,1	
47	0	0	0,0,40,1	0,9,46,1	0,10,28,1	0,26,44,1	9,1,1	26,3,1	10,2,1	
59	0	1	0,0,49,1	0,9,46,1	0,10,28,1	0,26,44,1	9,1,0	26,3,0	0,0,0	
62	9	1	0,0,49,1	0,9,46,1	1,0,61,1	0,26,44,0	0,2,1	26,3,0	0,0,0	26
64	26	1	0,0,49,1	0,9,46,1	1,0,61,1	1,9,64,1	0,2,1	9,3,1	0,0,0	
69	1	1	0,0,49,0	1,26,66,1	1,0,61,1	1,9,68,1	0,2,1	9,3,1	26,1,1	
74	0	1	1,1,71,1	1,26,26,1	1,0,61,1	1,9,73,1	1,0,1	9,3,1	26,1,1	
117	0	0	1,1,71,0	1,26,66,0	1,0,76,0	1,9,73,0	1,0,0	0,2,0	26,1,0	
120	9	0	0,0,119,1	1,26,66,0	1,0,76,0	1,9,73,0	0,0,1	0,2,0	26,1,0	
122	10	0	0,0,119,1	0,9,121,1	1,0,76,0	1,9,73,0	0,0,1	9,1,1	26,1,0	
123	26	0	0,0,119,1	0,9,121,1	0,10,123,1	1,9,73,0	0,0,1	9,1,1	10,2,1	
125	0	0	0,0,119,1	0,9,121,1	0,10,124,1	0,26,124,1	26,3,1	9,1,1	10,2,1	

Table legend:

- grey field - indicates IPT page fault in columns 4-7 and TLB miss in columns 8-10
- IPT[i] field - respectively: pid, vpn, lastUsed, valid
- TLB[i] field - respectively: vpn, phy, valid

Ticks analysis (only most informative ticks listed)

Tick 10
Process with pid = 0 starts to execute.

Initially, both the IPT and TLB are empty and valid bits are initialized to ‘false’. Therefore, TLB miss and IPT fault occurs what results in loading vpn value to IPT[0] and the record consisting of vpn and phy into TLB[0].

Tick 13
In this tick desired vpn is 9 so the procedure repeats as there are no matching entries in TLB. While IPT does not have corresponding entry, lruAlgorithm finds (in this case) first invalid record - IPT[1] which is then replaced.
Static variable – FIFOPointer points to the TLB[1] what results in updating this field with vpn 9 and phy 1.

Tick 15
Program looks for vpn 26 with pid 0. Since vpn is not present in both IPT and TLB procedure is the same as described under tick 10 and 13.

Updated fields: IPT[2] and TLB[2]

Tick 20
Program looks for vpn 1 with pid 0. Since vpn is not present in the TLB and IPT, lruAlgorithm finds invalid entry – IPT[3] which is loaded with desired values and TLB[1] is updated on the FIFO policy as all the TLB entries are valid already.

Tick 26
Program looks for vpn 0 within pid 0. In this case only TLB miss occurs because IPT[0] is already loaded with required vpn. As a result InsertToTLB function updates TLB[1] with value just found in IPT.

Tick 28
Program looks for vpn 10 with pid 0. In this tick TLB miss and page fault occurs. But due to the fact that IPT is already full, page 26 has to be paged out (as the one with the smallest number of ticks what indicates that it has not been used for longest period of time). On its place page 10 is paged in and then TLB table is updated.

For clarification: IPT is updated upon ‘last recently used’ policy. LastUsed value for IPT[2] is 17 (lowest value) what make that entry the most idle inside the IPT.

Tick 41

Required	vpn 9 with pid 0
Event	TLB miss
	There is no IPT fault as desired entry in present in IPT[1]
Result	TLB[0] is updated

Tick 42

Required	vpn 26 with pid 0
Events	TLB miss and IPT fault
Result	IPT[3] and TLB[1] are updated

Tick 47

Required	vpn 0 with pid 0
Event	TLB miss
	There is no IPT fault as desired entry in present in IPT[0]
Result	TLB[2] is updated

Tick 49

At this tick context switch occurred and thread ‘main’ with pid=0 was switched to thread ‘userprogram’ with pid=1.

Notice: for report clarity and to avoid text repetition just crucial details about each tick (from table) are shown below

Tick 59

Required	vpn 0 with pid 1
Events	TLB miss and IPT fault; process 1 executes
Result	IPT[2] and TLB[0] are updated

Tick 62

Required	vpn 9 with pid 1
Events	TLB miss, IPT fault, Paging out 26 page
	There is no IPT fault as desired entry in present in IPT[0]
Result	IPT[3] and TLB[1] are updated

Tick 64

Required	vpn 26 with pid 1
Events	TLB miss and IPT fault
Result	IPT[1] and TLB[2] are updated

Tick 69

Required	vpn 1 with pid 1
Event	TLB miss and IPT fault
Result	IPT[0] and TLB[0] are updated

Tick 74

Required	vpn 0 with pid 1
Event	TLB miss
	There is no IPT fault as desired entry in present in IPT[2]
Result	TLB[1] is updated

Tick 86

At this tick context switch occurred and thread called userprogram with pid=1 was switched to thread called main with pid=0. After that no longer needed thread ‘userprogram’ is deleted.
Therefore entries associated with ‘userprogram’ within both IPT and TLB tables become invalid (valid bit is set to 0).

Tick 117

Required	vpn 0 with pid 0
Event	TLB miss and IPT fault
Result	IPT[0] and TLB[0] are updated

Tick 120
Required vpn 9 with pid 0
Event TLB miss and IPT fault
Result IPT[1] and TLB[1] are updated

Tick 122
Required vpn 10 with pid 0
Event TLB miss and IPT fault
Result IPT[2] and TLB[2] are updated

Tick 123
Required vpn 26 with pid 0
Event TLB miss and IPT fault
Result IPT[3] and TLB[0] are updated

Tick 125
Required vpn 0 with pid 0
Event TLB miss
Result TLB[1] is updated

Tick 127
This is the last tick of the program. System call arises and machine halts what definitely terminates the 'main' thread.

Summary

Information derived from the code or debug output:

• Number of records in TLB	3	*machine.h*
• Number of records in IPT	4	*machine.h*
• Number of page faults	14	
• Number of page outs	2	
• Number of TLB misses	19	
• Total ticks	127	
• Page size (equals to SectorSize)	128 bytes	*disk.h*
• Size of the memory (128 Bytes*4)	512 bytes	

SectorSize was found in the nachos directories using linux "grep" tool.

Additional comments

Completion of the table above has been done after analysing the ‘DEBUG’ lines from the program.

```
// before updating memoryTable (IPT)
DEBUG('p', "Before updating IPT -----reading memoryTable ... \n");
for(int i=0;i<NumPhysPages;i++)
DEBUG('p', "Before paging out: IPT(i,pid,vpn,lastUsed,valid) IPT(%i,%i,%i,%i,%i) \n", i, memoryTable[i].pid,memoryTable[i].vPage,memoryTable[i].lastUsed,memoryTable[i].valid);
//update memoryTable for this frame
```

Above debug lines were implemented inside PageOutPageIn() function before IPT is updated.

```
// before inserting to TLB
for(int j=0;j<TLBSize;j++)
DEBUG('p', "Before TLB insertion: TLB(i,vpn,phy,valid) TLB(%i,%i,%i,%i) \n", j, machine->tlb[j].virtualPage, machine->tlb[j].physicalPage, machine->tlb[j].valid);
// IPT before updating TLB
for(int a=0;a<NumPhysPages;a++)
DEBUG('p', "Before inserting into TLB: IPT(i,pid,vpn,lastUsed,valid) IPT(%i,%i,%i,%i,%i) \n", a, memoryTable[a].pid,memoryTable[a].vPage,memoryTable[a].lastUsed,memoryTable[a].valid);
//update the TLB entry
```

These lines of code can be found in InsertToTLB() function, just before TLB entries are updated.

References

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, (2013) *Operating System Concepts*, Ninth Edition
- Lecture slides (Virtual Memory section)